AFRL-IF-RS-TR-2003-251
Final Technical Report
October 2003

THE EXPRESS PROJECT

Massachusetts Institute of Technology

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. E096

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

REPORT DOCUMENTATION PAGE	*Form Approved OMB No. 074-0188*

1. AGENCY USE ONLY (Leave blank)	**2. REPORT DATE** OCTOBER 2003	**3. REPORT TYPE AND DATES COVERED** Final Mar 97 – Sep 02

4. TITLE AND SUBTITLE THE EXPRESS PROJECT	**5. FUNDING NUMBERS** C - F30602-97-2-0013 PE - 62301E PR - E096 TA - 01 WU - 01
6. AUTHOR(S) Robert Laddaga, Olin Shivers, and Greg Sullivan	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Massachusetts Institute of Technology 77 Massachusetts Avenue Cambridge Massachusetts 02139	**8. PERFORMING ORGANIZATION REPORT NUMBER** N/A
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Defense Advanced Research Projects Agency 3701 North Fairfax Drive Arlington Virginia 22203-1714; AFRL/IFTB 525 Brooks Road Rome New York 13441-4505	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER** AFRL-IF-RS-TR-2003-251

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Mark J. Gorniak/IFTB/(315) 330-7724/ Mark.Gorniak@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.	**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(Maximum 200 Words)***

Military mission-critical software systems must continue to operate effectively in the presence of such failure modes as hardware failures, software bugs, and changes in the external environment. To enable such a capability, this effort proposed a general layered framework (the Dynamic Domain Architecture) for constructing adaptive software systems that can reason about the high-level goals of an application, monitor its fitness, diagnose problems, and reconstruct the application given a particular diagnosis. The system is aware of alternative approaches to achieving its goals, including alternative implementations of similar functionality, and debugging techniques for recovering from exceptions in its current behavior. It automatically and dynamically reconfigures the system and employs these alternative approaches to realizing its goals. In support of this framework, an extensible notation was developed that allows the programmer to annotate software with extra information: pragmatics, requirements, invariants, and metrics. Also a reasoning system for monitoring and diagnosis was developed, as well as abstraction tools for constructing software components in advanced programming languages.

14. SUBJECT TERMS Dynamic Domain Architecture, DDA, Self-Test, Functional Programming Languages, Embedded Software Development, Software Construction, Software Diagnosis, Repair, Rollback, Resourcing, Java, Dynamic Language Virtual Machine, DVM Technology, Athena, GRAVA	**15. NUMBER OF PAGES** 38
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

List of Figures and Tables

1 Project Objective

The objectives of the Express project at MIT have been to:

- close the feedback loop on software construction;
- produce tools for constructing highly adaptive software;
- make advanced, functional programming languages useful, and make practical tools for embedded software development, including serious systems programming;
- harness advanced reasoning tools to serve the above goals, with application of such reasoning during development and at runtime.

Closing the feedback loop requires monitoring and diagnosing software at runtime. It also requires software with adaptive capabilities. Making functional languages effective for embedded systems and systems software requires language improvements, type checking and compilation improvements, and aggressive, yet safe, optimization techniques. The first three goals all involve significant additional reasoning at development and runtime, to support more sophisticated program, language and application implementation techniques.

2 Overview

A typical modern-day software application is an enormous, monolithic system; and maintaining and evolving it is a Herculean task requiring large teams of engineers. Software systems change perhaps once or twice a year, involving a painful upgrade on the user's part—a process that can require hours of time and the interaction of a skilled technician.

This state of affairs will not support the demands of mission-critical field systems that operate in a changing, hostile environment. These software applications should not be rigid constructions that "shatter" when confronted with change; they must be flexible constructs that fluidly adapt to circumstance.

Military field applications of the future will be massively distributed systems, operating in an evolving environment; they will include modules that do not always function as desired. The presence of such failure modes will be unavoidable due to hardware failures, software bugs, and changes in the external environment. But mission-critical software systems must continue to operate effectively even in such changing circumstances.

The path to adaptability is through feedback: software that can reason about the high-level goals of the application, monitor its fitness, diagnose problems, and reconstruct the application given a particular diagnosis. Such a system should be aware of alternative approaches to achieving its goals, including alternative implementations of similar functionality, and debugging techniques for recovering from exceptions in its current behavior. In these cases, it should automatically and dynamically reconfigure the system and employ these alternative approaches to realizing its goals.

Embedded software should be developed against the backdrop of *frameworks*, each tailored to a single (or a small set of) issues. Some frameworks deal with cross-cutting issues such as fault tolerance; others deal with particular components and functional layers of a complex system (e.g. control of a sensor asset).

A framework, as we use the term, includes:

1. A set of properties with which the framework is concerned,
2. a formal ontology of the domain,
3. an axiomatization of the core domain theory,
4. analytic (and/or proof) techniques tailored to these properties and their domain theory,
5. a runtime infrastructure providing a rich set of layered services,
6. models describing the goal-directed structure of these software services,
7. a protocol specifying the rules by which other software interacts with the infrastucture provided by the framework,
8. and an embedded language for describing how a specific application couples into the framework.

A framework thus reifies a model in code, API, and in the constraints and guarantees the model provides. Frameworks should be developed with a principled relation to the underlying model, and preferably generated from a specification of the model. The specification of the model should be expressed in terms of an embedded language that captures the terms and concepts used by application domain experts.

The ontology of each framework constitutes a component of a semantically rich meta-language in which to state annotations of the program (e.g. goals, alternative strategies and methods for achieving goals, subgoal structure, state-variables, declarations, assertions, and requirements). Such annotations inform program analysis. They also facilitate the writing of high level generators that produce wrapper code integrating the multiple functional frameworks.

System development involves a new player, the framework developer, who plays the role of a bridge between application developers and systems programmers. Framework developers:

- provide tools that are too domain (or issue) specific for general system programmers to attend to, but too much in the style of core system code for application programmers to attend to;
- extend and raise the level of the language and infrastructure that the application programmer uses to solve problems;
- provide tools that synthesize the necessary low-level reactive code from the high level embedded language of the framework.

The core functionality of the system is decomposed along physical lines. Families of functionally similar components in a common domain (e.g. Optical Sensors) are managed by parameterized frameworks that cover that domain. Such a framework embodies the domain architecture for this area of functionality. A domain architecture structures the procedural knowledge of the domain into layers of services, each capable of achieving specific goals; a service at one layer invokes services from the lower layers to achieve its subgoals. Each service has many implementations corresponding to the variability and parameterization of the domain. Each alternative implementation represents an alternative strategy, method, or plan for achieving the goal. The choice of which implementation is to be invoked is made at runtime, in light of runtime conditions, with the goal of maximizing expected utility. Such frameworks are therefore *Dynamic* Domain Architectures. Each such framework exposes to other frameworks its models, goal structure, state-variables, its API, its protocol of use and constraints on those subsystems that interact with it.

Conceptually, these frameworks interact at runtime by observing and reasoning about one another's state and by posting goals and constraints to guide each other's behaviors. The posting of goals and constraints and the observation of state is facilitated by wrapper code inserted into the code of each framework at generation time by model-based generators of the interacting frameworks. The use of generated observation and control points, as well as the use of novel, fast propositional reasoning techniques allow this to happen within reactive time frames. The composite system behaves as if it is goal directed while avoiding the overhead normally associated with generalized reasoning.
The full application now consists of core functionality built by composing a variety of model-based *frameworks* as well as a variety of *cross-cutting* aspects, each written in the embedded language of a specific framework and each describing how that aspect is woven into the core functionality.

The frameworks capturing Domain Architectures as well as those capturing various crosscutting aspects evolve and are maintained separately; they are bound into the final system as late as necessary. Software maintenance and evolution are decomposed along the lines of frameworks.

A rich object-oriented language system, derived from the best ideas of Lisp and similar languages, can provide the base level platform for this new approach to embedded systems development. This core language system consists of a Virtual Machine, a Meta Object Protocol, a class system featuring multiple inheritance, multiple argument dispatch and method combination, a dynamic condition handling and recovery facility, and a powerful procedural macro system. These tools provide great power for expressing and synthesizing the code implementing model-based frameworks.

This rich Dynamic Object Oriented Programming environment provides these rich services not just in the development cycle but also within the runtime environment. The availability of runtime dynamic redefinition, late binding, condition and exception handling, generalized diagnosis and recovery support make this the idea platform for supporting not just DDA frameworks but also for supporting the model-based reactive executives.

2.1 Off Line Analysis

Although our focus is on the two areas above, it is important to understand that this approach makes the static analysis of the system easier. At both the Framework and the Executive level, such a system is inherently adaptive, attempting to guide itself away from anomalous states and back towards the intended behavior. This adaptivity makes the static analysis easier by reducing the possibility of the system taking excursions far away from the intended behavior.

In addition, the composition methodology itself lends itself to analysis. Part of each framework is a protocol specifying the rules under which other components are supposed to interact with the infrastructure provided by the framework. Corresponding to this protocol is a proof that certain properties will hold as long as the protocol is adhered to. This proof is conducted off line once by the framework developers and delivered as part of the framework. The fact that the protocol is adhered to by other parts of the system is often guaranteed by simple inspection or checking methods. Thus, part of the analysis of the system is done once off line, and then repeatedly reused.

Corresponding to each framework are analytic tools that can be used to examine the code that couples to this framework. Each such analytic framework can show that a set of properties in its area of concern is guaranteed to hold as long as the remaining code satisfies a set of constraints. Analysis of the system can proceed iteratively with each framework first showing that it satisfies the constraints placed on it by others and then determining what constraints it places on its sibling frameworks in order to guarantee its properties of interest. The analysis and understanding of the overall behavior of the system is, therefore, decomposed in just the same way as are the development, maintenance and evolutionary tasks.

Each framework establishes its own natural proof techniques; therefore heterogeneous reasoning capabilities are needed to support the analysis of software decomposed into frameworks.

3 Approach

Our approach has six basic technology components:

1. A general framework for constructing adaptive systems, the *DynamicDomain Architecture*.

2. A general, extensible notation that allows the programmer to annotate software with extra information: pragmatics, requirements, invariants, metrics.

3. A reasoning system for monitoring and diagnosis.

4. Abstraction tools for constructing software components in advanced programming languages.

5. A major thrust of the Express project was exploring the use of functional programming languages, such as Scheme or SML, for systems programming. This began with Scsh, a Unix systems-programming environment embedded within the Scheme programming language [Shi94].

6. A proof-based language that is formally well grounded but as natural to use as modern programming languages.

We are developing frameworks for more expressive programming---allowing the programmer to write down facts about his program in a notation that is amenable to machine processing. Moving this information out of the comments and into the program gives software tools access to the information, for debugging, optimization, static checking, and flexible reconfiguration of large software systems. There are two key challenges we must address here:

1. How to provide these features in an extensible framework that can be adapted to the particular demands of a given task domain.

2. The design of a notation for expressing support for propositions that can be mastered and used successfully by practicing software engineers, rather than by professional logicians.

Our approach to building adaptive software is based on the notion of a Dynamic Domain Architecture (DDA). Dynamic Domain Architectures structure an application domain into layers of common services where each service has a number of variant implementations tailored to different environmental conditions. The architectural level of description also provides "purpose links" which explain how the components of a service achieve its overall goals. The DDA development environment synthesizes run-time sentinels to monitor the preconditions of the purpose links. The runtime services of the DDA are invoked when a sentinel signals the failure to achieve an expected condition; the runtime services are responsible for diagnosis of the failure and for selection and execution of a repair procedure. Since the DDA provides many alternative implementations of each component, a typical repair involves rolling back to a recovery point and invoking an alternative implementation.

We are generally exploring program analyses to allow powerful, expressive programming features to be used efficiently to construct real systems programs. Our approach exploits the solid formal underpinnings provided by advanced functional languages, as well as the analytic power provided by proof annotation.

3.1 Milestones

3.1.1 DDA Milestones

Task 1.0	Develop and extend Scsh, a Unix systems-programming environment embedded within the Scheme programming language. Use of functional programming languages, such as Scheme or SML, for systems programming.

Task 2.0	Develop technology for transducer networks that do not have a simple linear “pipeline” data-flow topology.
Task 3.0	Design and Development of a programming language for DDA. In this task we will use the lessons learned from our earlier experiences to design and develop a Dynamic Object Oriented Programming language in the Lisp tradition which is tailored to the new programming paradigm developed in the earlier work.
Task 4.0	Application Demonstration. In this task we will conduct a modest demonstration of the new language system by picking a prototype application and implementing it in the new system.

3.1.2 Off Line Analysis Milestones

Task 1.0	Develop a proof based language that is formally well grounded but as natural to use as modern programming languages.

3.1.3 DOLL Subcontract Milestones

Task 1.0	Build additional agents
Task 1.1	Test extended system on images
Task 1.2	Report on the results of self-adaptation to robustness and stability of visual interpretation.
Task 2.0	Design the hooks and tools required for debugging and monitoring of self-adaptive programs based on the developed architecture.
Task 2.1	Implement the hooks and tools designed in task 2.0
Task 2.2	Document and report on the self-adaptive program design, development, debugging and monitoring paradigm developed under this program.
Task 3.0	Produce a plan for a generalization of the self-adaptive agent architecture by looking at applying the architecture to a very different

	problem domain. Ideally this will be done in collaboration with a partner that we will attempt to identify as a consumer of this technology.
Task 3.1	Implement and document the generalization of the architecture designed in Task 3.0.

3.2 Outline

Although the motivation for this project is a comprehensive embedded systems development environment, the investigation of individual component technologies was generally conducted in individual environments. The remainder of this report documents the individual subprojects and milestones of the Express / DDA project.

4 Dynamic Domain Architectures

A Dynamic Domain Architecture structures a domain into service layers; each service is annotated with specifications and descriptions of how it is implemented in terms of services from lower levels. Like other domain architectures, a Dynamic Domain Architecture provides multiple instantiations of each service, with each instantiation optimized for different purposes. Thus, it serves as a well structured software repository. Typically, the application is a relatively small body of code utilizing the much larger volume of code provided by the framework. Typical domains of concern for military embedded software systems include sensor management, navigation guidance and control, electronic warfare, etc.

A Dynamic Domain Architecture is, however, different from the domain architectures developed in earlier DARPA programs (e.g. STARS and DSSA). In earlier systems, the Domain Architecture was a static repository from which specific instantiations of the services were selected and built into the run-time image of the application. Neither the models nor the deductions used to select specific instantiations of the services are carried into the runtime environment. In a Dynamic Domain Architecture, however, all the alternative instantiations, plus the models and annotations describing them are present in the run-time environment, and multiple applications may simultaneously and dynamically invoke the services.

Dynamic Domain Architectures allow late binding of the decision of which alternative instantiation of a service to employ. Like Dynamic Object Oriented Programming (DOOP) systems, the decision may be made as late as method-invocation time. However, Dynamic Domain Architectures go further than DOOP, allowing the decision to be made using much more information than simple type signatures. The models which describe software components are used to support runtime deductions leading to the selection of an appropriate method for achieving a service.

Dynamic Domain Architectures recognize that in many open environments (e.g., image processing for ATR) it isn't possible to select the correct operator with precision, *a* *priori*. Therefore, Dynamic Domain Architectures support an even later binding of operator selection, allowing this initial selection to be revised in light of the actual effect of the invocation. If the method chosen doesn't do the job well enough, alternative selections are explored until a satisfactory solution is found or until there is no longer any value to be gained in finding a solution.

Dynamic Domain Architectures remove exception handling from the purview of the programmer, instead treating the management of exceptional conditions as a special service provided by the run-time environment. The annotations carried forward to run time include formal statements of conditions which should be true at various points of the program if it is to achieve its goals. The DDA framework generates runtime monitoring software that invokes error-handling services if these conditions fail to be true. The exception-management service is informed by the Dynamic Domain Architecture's models of the executing software system and by a catalog of breakdown conditions and their repairs; using these it diagnoses the breakdown, determines an appropriate scope of repair and possibly selects an alternative to that invoked already; it then restarts the computation.

Finally, a DDA framework provides an embedded language in which the developers of other frameworks can access its state-variables and influence its goal and plan structure.

4.1 Domain modeling

The idea of domain architecture dates back to the Arpa Megaprogramming initiative where it was observed that software reuse could best take place within the context of a Domain Specific Software Architecture. Such an architecture would identify important pieces of functionality employed by all applications within the domain, and would then recursively identify the important functionality supporting these computations. In a visual-interpretation domain, for example, typical common functionality might include region identification, which in turn depends on edge detection, which in turn depends on filtering operations (eg, convolutions).

This process of identifying and structuring the common functionality is the first component of a process termed Domain Analysis. Domain Analysis structures common functionality into a series of "service layers," each relying on the ones below for parts of its functionality. The second component of Domain Analysis is the identification of variability within the commonality. Returning to our visual-interpretation example, there are several different approaches to region identification, dozens of distinct edge-detection algorithms, and many different ways to perform filtering operations. When looked at in even finer detail, there may be an even greater number of variant instantiations of any of these operations. Variations arise due to different needs for precision, time and space bounds, error management, and the like.

The power of Domain Analysis is that its identification of common functionality lets one view the code in new terms: the bulk of the code is in the service-layer substrate and implements functionality common to many applications. Each application consists of a thin veneer of application-specific code, riding on top of this substrate of service layers. However, the substrate contains many variant instantiations of each service. Although each instantiation is relevant to only some of the applications, Domain Analysis lets us see these as variants of a common conceptual service. Before the Domain Analysis was performed, each application stood alone using its particular instantiations of the common services and was ignorant of the fact that other applications used the same conceptual services but with different instantiations.

4.2 Dynamic Object Oriented Programming

The Lisp community, with its close connection to artificial intelligence research, has independently discovered some of the same ideas, but has packaged them in a more dynamic but less formal framework. This approach was first identified and termed "super-routines" in a paper by Erik Sandewall [San79, SSS81]. Sandewall noted that it is often the case that a whole class of computations, are instances of some very general pattern of computation, where the members of the class differ only in the details. He termed this higher-level structure a "super-routine" and noted that data-driven programming techniques could dynamically determine which subroutine was relevant at run time.

As object-oriented programming ideas developed in the Lisp and Smalltalk communities, several researchers began to understand that the Dynamic OOP facilities common in these languages were exactly what is needed to build a super-routine.

The high-level common services of a Domain Architecture are precisely the same idea as Sandewall's notion of a super-routine (rediscovered in another context by another community a decade later). Unlike the Static Domain Architectures, the runtime environment of the systems Sandewall characterized all included many variant instantiations of the common services, and dynamically invoked a particular instantiation based on run-time conditions.

The mapping between the super-routine (or Domain Architecture) idea and the features of DOOP is straightforward: each high-level abstract operation (or Domain Architecture service) is identified with a generic function; the different instantiations are provided by different methods, each with a unique type signature. Method invocation performs the dynamic run-time selection of the appropriate instantiation of the service. This style of building extensible, domain specific architectures has become known as "open implementation" [Kic96].

Dynamic OOP also provides significant facilities for managing exceptional conditions. In the case of Lisp, these facilities were motivated by the needs of adaptive planning systems. In particular, the facilities provided to signal exceptional conditions allow the error-handling code access to the environment of the exception and this, in turn, allows the handler to characterize the nature of the breakdown. Facilities similar to the signaling of the exception are used to transfer control from the error handler to an appropriate "restart" handler. Once the error handler has characterized the nature of the breakdown, it invokes a repair mechanism, not by name, but by description.

Finally, the language provides facilities to specify what cleanup work must be done to perform the appropriate recovery work as rollback to the restart position takes place. We enrich this infrastructure with extensive models of the software's structure, function and purpose and to build in facilities for noticing if an operator has failed to achieve its purpose. Then we must carry into the run-time environment all the descriptive information as well as all the variant instantiations of operators present in the development environment, and we use this information reactively to control the physical system in which the software is embedded. We call this type of framework a Dynamic Domain Architecture because it incorporates and extends ideas from the two traditions of Domain Architectures and Dynamic Object Oriented Programming.

4.3 Services Provided

A Dynamic Domain Architecture is far more introspective and reflective than conventional software systems. This allows many tasks which currently burden the programmer to instead be synthesized from the models within a framework or to be provided as system services.

Figure 1 shows a schematic of the monitoring, diagnosis and repair processes. Failure to achieve a pre- or post-condition triggers the diagnostic service, which eventually results in a concrete repair plan, which we resource and then execute.

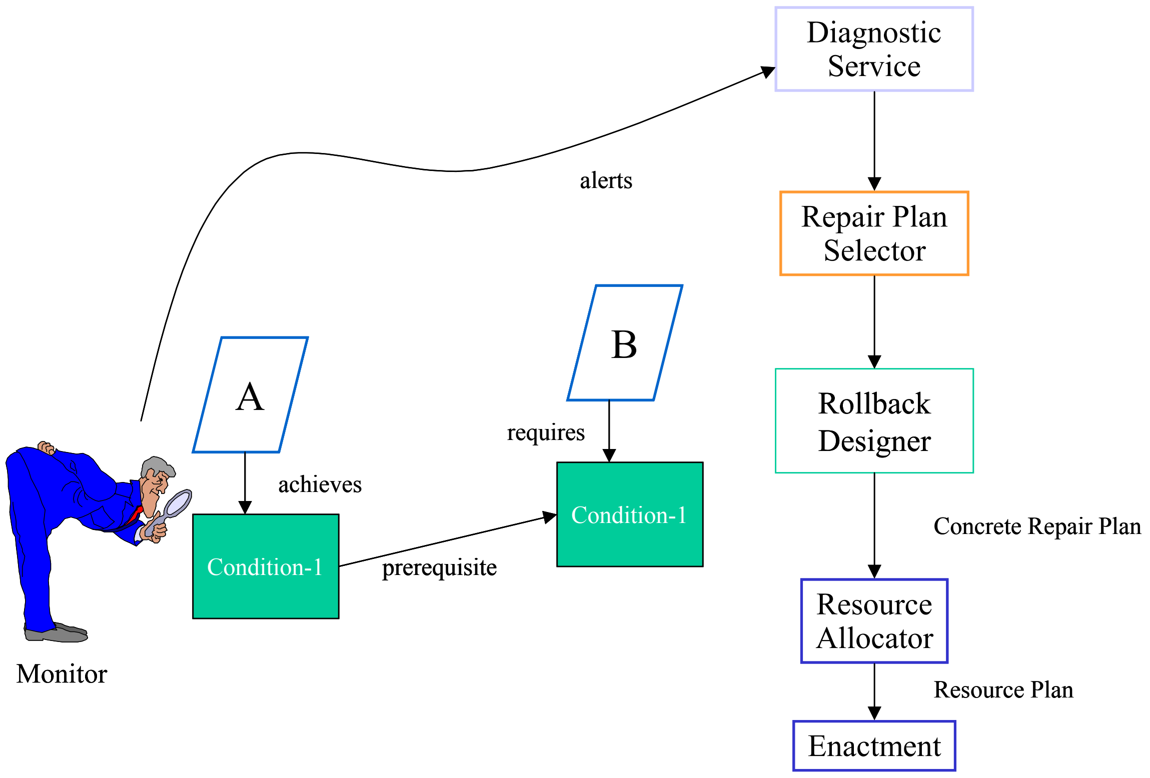

Figure 1. Making the System Responsible for Achieving Its Goals

A more complete description of the services in the DDA is shown in Figure 2. These include:

1. The synthesis of code that selects which variant of an abstract operator is appropriate in light of run-time conditions.

2. The synthesis of monitors which check that conditions expected to be true at various points in the execution of a computation are in fact true.

3. Diagnosis and isolation services which locate the cause of an exceptional condition, and characterize the form of the breakdown which has transpired.

4. Alternative selection services which determine how to achieve the goal of the failed computation using variant means (either by trying repairs or by trying alternative implementations, or both).

5. Rollback and recovery services which establish a consistent state of the computation from which to attempt the alternative strategy.

6. Allocation and re-optimization services which reallocate resources in light of the resources remaining after the breakdown and the priorities obtaining at that point. These services may optimize the system in a new way in light of the new allocations and priorities.

7. The synthesis of connections to reactive executives that manage physical components of concern to the DDA framework.

8. The synthesis of connections to other DDA frameworks with whose state-variables, goals and plans the current framework interacts.

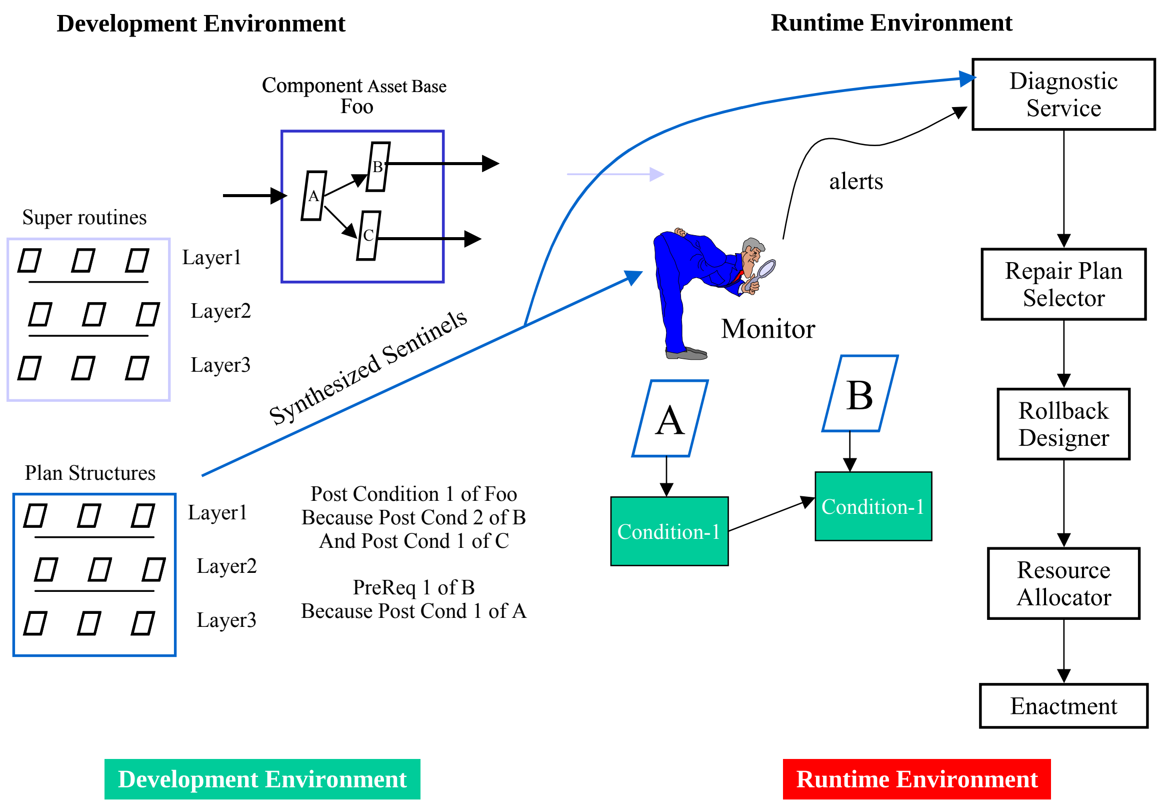

Figure 2. Dynamic Domain Architecture Services

Development Environment	**Runtime Environment**
Captures Super-routine layers	Error Signaling
Captures Plan Structures	Diagnostic Service
Instruments Code with Condition Signalers & Handlers	Recovery and Rollback
Instruments Code with Restarts and Unwind Protects	Resource Reallocation
Identifies needs for extra resources and/or redundancy	Restarting Computation

Table 1. Development and Runtime Services of the DDA

4.4 Key Components

We identified the key components of a Dynamic domain architecture and studied how the pieces interact. We have implemented several components of the overall vision: We have created a prototype interactive development environment which facilitates the capture of both code and annotations. In particular, we have a natural way of capturing "purpose" annotations which explain how one component of the system relies on another. Purpose annotations are the raw material from which runtime monitors will be synthesized. We implemented a model-based troubleshooter component for our Dynamic Domain Architecture system. The troubleshooter is driven by a "plan description" of the system, an abstract description including decomposition, data and control flow links and constraints on the components' behaviors. In the current version we have concentrated on quality of service descriptions, such as expected delay times. The troubleshooter uses a novel multi-layered Bayesian representation that allows it to make inferences not just about the failure modes of the computation, but also about the likelihood of failures in the underlying infrastructure.

We have designed and implemented a transactional memory system as part of a byte code emulator for a dynamic language (common lisp). We have designed and implemented a "dynamic language virtual machine" (DVM). The DVM is a very general engine capable of supporting many languages (Java, Common Lisp, Dylan); it includes a complete meta-level along the lines of the CLOS MOP (Meta Object Protocol).

5 Dynamic Language Extension Efforts

In this section we provide detail on subprojects to extend dynamic language capabilities, to better serve the goals of the DDA.

5.1 Macros for Java

The ability to extend a programming language with new constructs is a valuable tool. With it, system designers can grow a language towards their problem domain, enhance productivity and ease maintenance. We have developed an extension to the Java language that allows Java programmers to define new syntactic constructs. The design is based on the Dylan macro system (e.g., rule-based pattern matching and hygiene), but exploits Java's compilation model to offer a full procedural macro engine. In other words, syntax expanders may be implemented in, and so use all the facilities of, the full Java language. The system is implemented and working as a Java preprocessor.

A talk on the Java Syntactic Extender was delivered at the ACM conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA'01) in Tampa Bay Florida in October, 2001 [BP01]. This work was also presented at an invited talk at BBN in February 2002.

5.2 Efficient Predicate Dispatching for Dylan

Predicate dispatch [EKC98] provides a generalization of other method dispatch techniques through the utilization of arbitrary predicates to control method applicability and logical implication between predicates to determine method overriding. It generalizes previous object-oriented single and multimethod dispatch, ML-style pattern matching, predicate classes, and classifiers. An efficient predicate dispatch implementation technique [CC99] involves reducing general predicate dispatch into multidispatch using a canonicalization process, mapping multidispatch onto a sequence of single dispatches through the construction of a decision DAG, and finally implementing single dispatch in terms of a binary search. Dylan provides a rich set of built-in types and a powerful multimethod dispatch mechanism that present several challenges to the predicate dispatch mechanism and its implementation. We have developed a mapping of Dylan types onto predicate types, a dynamic x86 code generator, and several improvements to the general implementation strategy.

5.3 Predicate Dispatching for CLOS

We have added predicate dispatching to the Common Lisp Object System (CLOS), using the metaobject protocol facilities of CLOS [Uck01]. We then demonstrated the utility of this enhancement by using predicate dispatching to extend a computer algebra system. The predicate dispatching facilities allowed us to implement algebraic operations more concisely than we would have been able to in the original system.

5.4 The Dynamic Virtual Machine

Partial evaluation is a technique to specialize application code with respect to values of some of the free variables in the code. Traditional partial evaluation is done at compile time, although there is a variant called "runtime partial evaluation" that defers some of the specialization to runtime. Dynamic partial evaluation [Sul01] goes much further, deferring all partial evaluation analysis and specialization to runtime.

The DVM has been extended with a Dynamic Partial Evaluator (DPE) which builds optimized versions of a method while evaluating it. The partially evaluated methods are themselves just methods with a more specific type signature. In our implementation, the new method is dynamically added to the system making it available for method dispatch in future runs. Thus the system dynamically optimizes itself as a byproduct of execution.

In 2000, we completed a second version of the design and implementation of our “Dynamic Language virtual machine” (DVM). We have extended the DVM to use the extremely general object model of Predicate Types, as described in Section 5.2.

We are implementing a translation from the Java programming language to the DVM. While Java is not especially well-suited for highly dynamic, adaptive software implementation, this implementation will give us a starting point for performance analysis. Also, we will be able to start adding features to Java that tap some of the more dynamic aspects of the underlying DVM.

5.4.1 GLOS

In lieu of a full Dynamic Virtual Machine, we have some extensions to Scheme, collectively known as the Generic Little Object System (GLOS). GLOS provides inheritance, multiple dispatch, predicate types, and a simple yet powerful metaobject protocol for method dispatch. Using the dispatch protocol, we provide multiple dispatch and method combination.

The features of GLOS were used in an experiment comparing implementations of the “Gang of Four” Design Patterns in C++ (as presented in the GOF book) and in GLOS. Using the advanced features available in GLOS allowed for more concise, more dynamic, and more “first class” implementations of many of the design patterns. This project is documented in AI Lab Memo 2002-005 [Sul02].

The library of Scheme utilities known as GLOS (Generic Little Object System) has been ported to the PLT implementation of Scheme, out of Rice and Northeastern Universities.

5.5 The GOO (née Proto) Programming Language

In 2000, a new language called Proto was invented. Proto is a prototype-based prefix syntaxed language that is meant to be simple, powerful and extensible. It is designed to provide both a research and teaching framework for pushing the limits of abstraction and dynamism and efficient and reliable delivery all while maintaining an ultra simple design and implementation. The implementation includes a just-in-time whole-program compilation including a dependency-tracking scheme that supports full interactive redefinition of highly optimized code and objects. Finally, the entire language is documented in less than ten pages and the implementation weighs in at less than ten thousand lines of code.

Core Proto is fully implemented including a suite of libraries and an interpreter. Proto is written almost entirely in itself, and is bootstrapped using a Proto-to-C compiler. Proto was used to teach a graduate-level seminar in advanced object-oriented dynamic language design and implementation techniques.

In 2001, the Proto language was renamed GOO to match the fact that the language is no longer prototype-based. A large amount of time was spent in actual language design. The overriding goal is to design an ultra simple and elegant language that is amenable to simple and effective optimizations while not sacrificing interactivity. An extensible lightweight dynamic type system was designed in such a way as to conveniently convey programmers' intent while at the same time providing type information to the optimizers and warnings to developers. A key innovation is an extensible parameterized dynamic type framework that smoothly integrates into multimethod dispatch by way of a unification mechanism.

Much work was spent on tuning the GOO runtime facilities and led to the development of a new fast subtyping representation, called the Packed Vector Encoding, that is extremely fast to construct, is one page of code to implement, and which is competetive with the best algorithms on a wide range of real-world class hierarchies. This work along with a general overview of GOO research was presented at the Lightweight Languages 1 (LL1) conference at MIT in November 2001.

The dynamic compiler infrastructure has been implemented including a general dependency-tracking framework. Interactive top-level interactions are implemented in terms of a dynamic compiler. We presented the general simple dynamic compilation architecture at the Harvard Computer Science department in February 2002.

An initial release of GOO was made on www.googoogaga.org in April 2002.

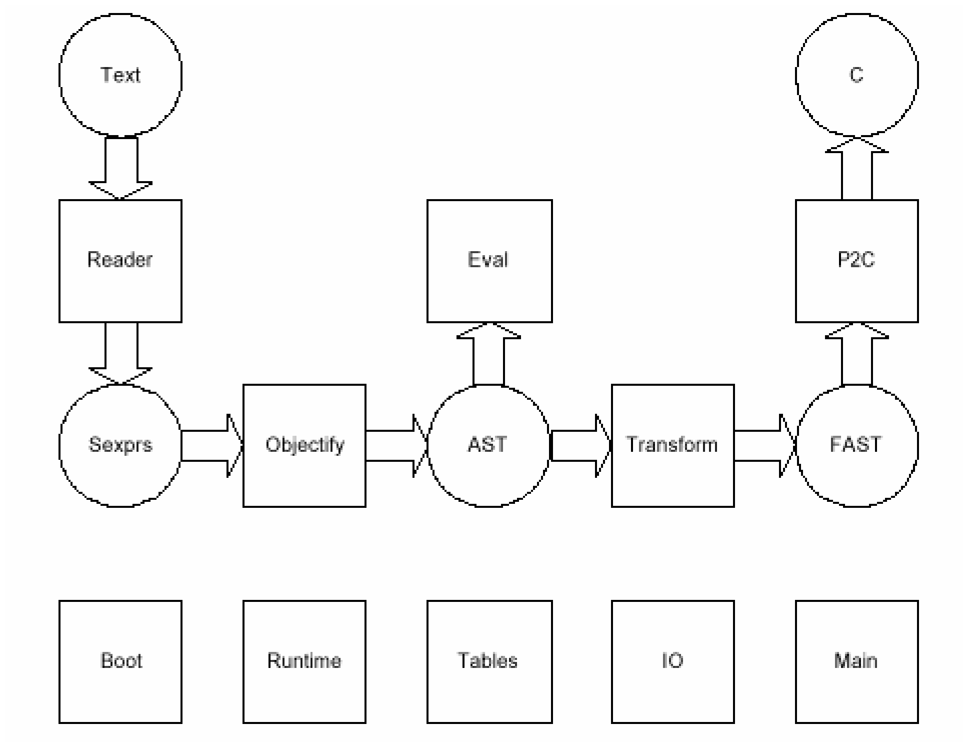

Figure 3. GOO Architecture

A key innovation is an extensible parameterized dynamic type system, developed with a master's student named James Knight. The focus of his work has been to add a parameterized type system to GOO that works with GOO's multimethod dispatch, and is as dynamic and extensible as possible. This research is based on the substantial amount of prior work in this area, but improves on what has already been done in its flexibility and usefulness. The goal was to create an extension of the type system that would enhance its expressiveness beyond that of the usual parameterization systems found in C++, ML, and Dylan by allowing the user to express covariant and contravariant relationships, when they exist.

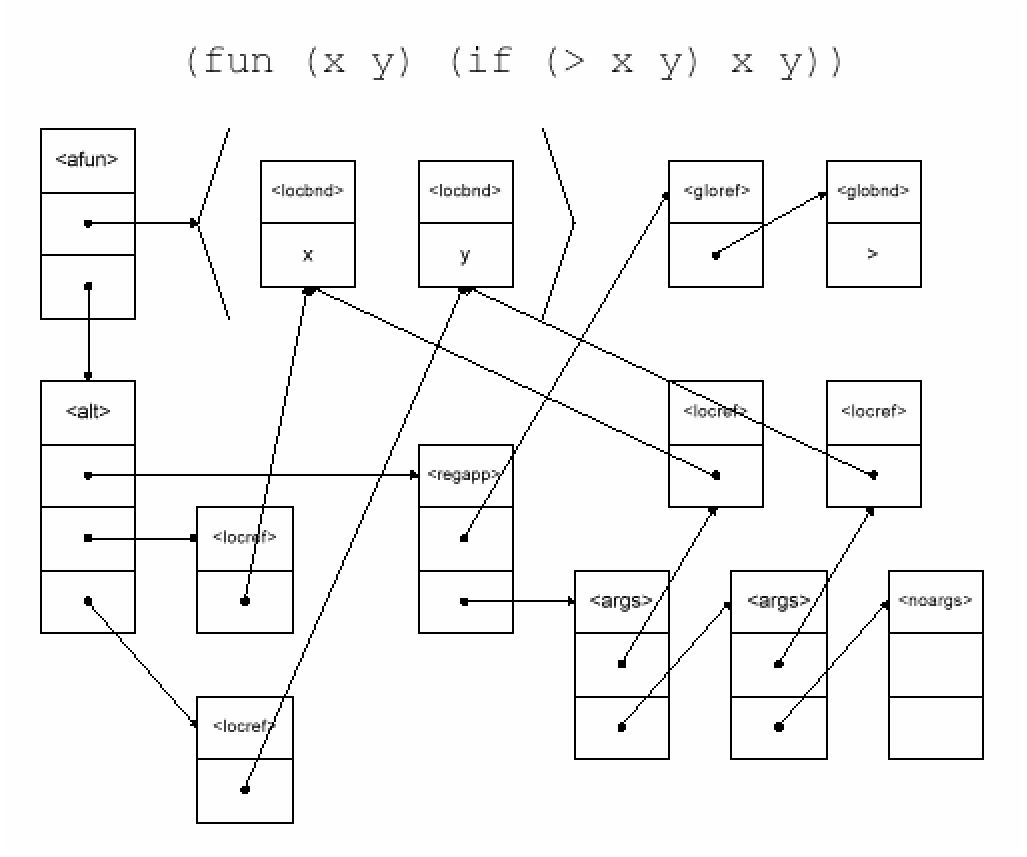

Figure 4. Example Abstract Syntax Tree in GOO

A new mechanism for implementing multimethod dispatch has been implemented on top of GOO's dynamic compilation infrastructure. The basic strategy is to translate generic function dispatch into a decision tree composed of subtype tests. Method bodies can be inlined into the tree and the tree itself can be inlined into call sites. These trees can then be pruned given available type information.

A threading facility has been added to GOO and GOO's runtime has been augmented to be mostly thread safe. The thread library is a lightweight layer over the portable Posix-standard pthreads API.

Much effort has been put into fostering a user community and open source development effort around GOO. GOO's documentation has been made available in HTML format, a bug database was installed, a WIKI has been created and developed, and a majordomo mail list has been created and archived.

5.6 Dynamic Optimization of Interpreters

With additional funding via a joint MIT and Hewlett-Packard alliance, we have been investigating the application of dynamic native optimization to the domain of interpreters for dynamic languages. Traditional JIT's ("just in time" compilers) or native-to-native dynamic optimization systems are confounded by interpreters. This is because the "hotspots" that are identified and optimized are the loops in the *interpreter*, rather than in the application being *interpreted*. If the interpreter can supply to the dynamic optimization system some notion of a "logical PC (logical Program Counter)", then the optimization system should be able to do a better job of identifying actual hot traces in the running system.

As of the end of the Express contract, no publishable progress was made in this direction, but promising research continues in this area.

6 Applying Functional Techniques to Systems Programming

A major thrust of the Express project was exploring the use of functional programming languages, such as Scheme or SML, for systems programming. This was a necessary effort in service of the application of DDA to embedded systems and systems programming in general. This work began with our work on scsh, a Unix systems-programming environment embedded within the Scheme programming language [Shi94]. Scsh remains the most complete Unix systems-programming environment developed to date in a functional language, and has been downloaded off the net and used for a wide variety of tasks: financial analysis, VLSI design, web servers, web clients, distributed systems development, database interfaces, and systems administration. We have continued to develop and extend scsh over the lifetime of the Express project, most recently releasing version 0.6.3 in January of 2003, found at http://www.scsh.net.

Besides serving as a platform for systems programming in Scheme, scsh also provided an opportunity to explore the use of embedded-language development for task-specific notations in systems program. The Scheme macro system allowed us to develop multiple such notations, for tasks such as process creation, pattern matching, and string processing. This leads to a particular paradigm for "domain-specific languages" that allows tight integration of multiple, distinct task-specific notations, all connected together by the powerful "glue" of the Scheme programming language. [Shi96b].

Programming languages and operating systems are artifacts with something in common: they are both specifications of a virtual machine. It could be said that Unix and C are symbiotic with respect to this commonality, in that Unix provides the fundamental run-time services needed by the C ``machine" model. This leads to a pair of related questions:

- How best to express these standard OS services in a manner that is most harmonious with the machine model presented by a functional language?

  The development of scsh allowed us to address this issue, with consequent gains in simplicity and expressiveness over the interfaces exported from the C model. As one example, we were able to provide "GC-like" automatic management for operating-system resources such as processes, asynchronous signals, and I/O handles [Shi97b]. Just as with automatic management of memory, allowing the automatic management of these resources in scsh provides gains in clarity, simplicity, modularity, and robustness of systems programs---completely eliminating errors such as creating zombie processes, or creating deadlocks by failing to release process I/O resources. Thus, functional languages serve the needs of systems programming.

- What are the fundamental OS services that are symbiotic with the needs of advanced functional languages?

  Our answers here proceeded from our years of work on scsh *within* the standard OS paradigm. To try them out, we constructed ML/OS, an implementation of Standard ML that ran on bare hardware---the language, in this case, *was* the OS. The resource-management, asynchronous exception, scheduling, and protection mechanisms are all those of the functional language. This was accomplished via the DARPA-funded OS Kit, developed by our colleagues at the University of Utah [FBB+97].

  One of the deeper connections between functional languages and operating systems lies in the realm of modelling processes and process state using continuations in Scheme. This opens up the possibility of exposing the patterns of thread interaction to language-level optimization [Shi97a].

  A final result of our work in ML/OS stemmed from the realization that the standard techniques for implementing storage allocation in a garbage-collected system interact very badly with thread schedulers. When an operating system's core scheduler is just some functional language's thread system, this is a serious issue … but it also surfaces in highly-threaded, high performance transaction systems built in languages like Java (something that has become quite common since we did this work). We designed a novel storage-allocation technique to address this issue in ML/OS, which provides both extremely low-latency context-switch times and extremely low overhead allocation costs [SCM99].

Continuing the theme of efficiently coupling the requirements of dynamic programming languages with the underlying resources of the target machine, we developed techniques to support the special needs of dynamic languages on the Java VM [Shi96a].

The common thread of this work is the notion of putting the powerful tool of advanced functional programming languages to work in a systems context. Only by "stressing" these languages in serious use can we discover their benefits, and, more importantly, their shortfalls, in order to gain understanding for further design.

6.1 Transducer Composition

We discovered a continuation-based technique for connecting computational elements represented as on-line transducers. This technique exposes the control- and data-flow of composed transducers to analysis and optimization using standard control-flow analysis approaches. The potential here is to “fuse” or optimize across compositions of elements such as network-protocol layers, DSP modules, iterators, or processing steps in high-performance graphics rendering algorithms.

The basic technique hinges on a CPS (Continuation Passing Style) program representation, fitting in with the general thrust in Express of applying the technology of advanced programming languages to real systems applications.

Over 1998, we generalized this technique to transducer networks that do not have a simple linear “pipeline” data-flow topology. We also implemented the analyses and optimizations in the SML/NJ compiler, developed by our collaborators at Bell Labs. This research is documented in a 1999 technical report, [Shi99].

7 ``Proof engineering'' and the Athena proof language

Proof engineering and the Athena proof language improve and extend the static analysis capabilities of the DDA.

7.1 Background

In the very early stages of the project, we constructed a Scheme interpreter that allows the programmer to annotate arbitrary expressions with assertions drawn from a logic defined in a first-order, Horn-clause (Prolog-like) system. These assertions are checked before executing the program. We have successfully implemented some simple “task-directed logics” using this prototype tool, such as dimensional analysis.

We experimented with HOL, a significantly more powerful theorem-proving system. Proof-checking in HOL can be carried out very efficiently, being essentially equivalent to type-checking. The richer logics supported by HOL should also allow us to represent more sophisticated analyses and annotations.

After spending some time with the proof systems such as HOL and LF, we launched into an ambitious project to develop a proof-based language that is formally well grounded but as natural to use as modern programming languages.

7.2 Athena-0

In 1998, we completed the design, supporting theory, and initial implementation of Athena, a new proof language. Athena is a novel departure from standard proof languages, especially those based upon the “types are theorems” Curry-Howard isomorphism, such as LF. In Athena, a proof is a program; evaluating this program produces a theorem (or an error, if the program makes an invalid step). The semantics of the language make it impossible to produce a false “theorem.”

The design of Athena has been driven by a dissatisfaction with the current notations for expressing theorems and proofs. While notations such as LF are quite general, they are not well-suited for use by humans---they are the logical analog of programming in assembly language. The Athena language exploits well-understood principles of language design, such as scope, abstraction, and higher-order values, to produce a compact, expressive notation for proving theorems.

The initial system, Athena-0, was an initial exploration of the basic proof techniques embedded in a dual computation/deduction linguistic setting. It provided support for proofs of arbitrary theorems in propositional logic. The language has a complete, formal specification, using operational semantics. There are several theorems establishing fundamental properties of the language, such as its soundness: if an Athena proof establishes a proposition, that proposition is guaranteed to be valid, i.e., it is a theorem.

7.3 Athena-1

In 1999 we developed the second version of Athena. Athena-1 extended the basic notation to predicate logic. Both of these designs have been performed with solid theoretical underpinnings: a complete formal semantics for the languages, along with associated proofs of correctness. Work has begun to produce printed tutorials and reference manuals to enable external users to use Athena. A recent, DARPA-funded research project has used Athena to construct a compiler that produces machine code along with a verifiable proof that the machine code is a faithful and correct translation of the program source.

7.4 Denotational Proof Languages

Several new Deductive Procedural Languages (DPLs) have been designed and implemented. These include type-ω DPLs [Ark01d] for classical logic and for various modal and temporal logics. New theorem-proving paradigms based on higher-order proof continuations have been discovered in DPL formulations of modal logics in connection with the necessitation rule. These paradigms enable one to write powerful theorem provers for so-called "normal" modal logics (such as T, S4 and S5, etc.) in a fluid and succinct style, and with a strong soundness guarantee.

DPLs introduce novel semantic abstractions and versatile mechanisms for constructing certificates that are not available in previous frameworks, such as logic-programming languages or theorem provers of the LCF/HOL variety. Athena, in particular, has already been used to express several sophisticated algorithms (such as the Hindley-Milner type-inference algorithm) as certificate-producing theorem provers. If this can scale to real-life programming, it will open up a whole new way of writing software.

7.5 Applications of DPL's to Concurrent Systems

In 2001, we focused on the specification and verification of complex concurrent systems. In particular, we have been exploring the construction of DPL theorem provers for several of the various temporal logics that have been found useful for specifying and reasoning about the behavior of concurrent systems, both linear and branching. We have worked on the logic used by Manna and Pnueli, Lamport's temporal logic of actions, and computation tree logic. We are also exploring the incorporation of a DPL-based theorem prover for first-order logic into IOA, a language for specifying concurrent systems using Lynch's formalism of input-output automata. Finally, we have been investigating the integration of model-checking techniques for finite case-analysis reasoning into type-ω DPL's.

7.6 Certified Computation

One of the most exciting new applications of sophisticated Denotational Proof Languages (DPLs) is what we have dubbed *certified computation*. A certified computation not only produces a result *r* but also a correctness certificate, which is a formal proof that *r* is correct. This can greatly enhance the credibility of the result: if we trust the axioms and inference rules that are used in the certificate, then we can be assured that *r* is correct. In effect, we obtain a **trust reduction**: we no longer have to trust the entire computation; we only have to trust the certificate. Typically, the reasoning used in the certificate is much simpler and easier to trust than the entire computation.

Certified computation has two main applications: as a software engineering discipline, it can be used to increase the reliability of our code; and as a framework for cooperative computation, it can be used whenever a code consumer executes an algorithm obtained from an untrusted agent and needs to be convinced that the generated results are correct.

We have been exploring the theme of certified computation in an attempt to build dataflow analyzers and smart compilers that prove their results. This could lead to a completely open, general, and extensible compiler architecture. For instance, an arbitrary source could contribute a new optimizing module, which we could trust and confidently use because it would justify its results. The basic idea is simple: programmers often know a lot of things about the behavior of their code. If the compiler knew these things too, it would often be able to improve the quality of the generated code by leaps and bounds. Some current smart compilers discover some of this knowledge on their own, but the task is usually hopeless. In fact it is provably impossible for the compiler to extract all the potentially useful information on its own---the problem is undecidable. Our proposal is to turn to the programmers. They already know all the answers, and usually they can back them up as well. Thus the idea is for the programmers to give information to the compiler. They tell the compiler: "Such and such is always true at this point in the program. If you don't believe me, here is a proof." The compiler will then check the proof, and if the proof goes through it can then go ahead and try to do some good things with the code based on the supplied information that the programmer provided. This idea is originally due to Knuth, and dates back to the early 1970s. It is lamentable that it was never fully pursued. We believe that DPLs have raised the state of the art in proof technology sufficiently high for this ambitious project to finally have realistic chances of success.

For a more detailed exposition of the application of DPL technology to certified computation, see [Ark01a].

7.7 Efficient Implementation of DPL's

We have also worked on efficient implementations of DPL’s. A graduate student in our group has defined and implemented a virtual machine for the monomorphic subset of Athena that extends the SECD abstract machine [Lan64] with various new constructs designed specifically for DPL features such as assumption bases. Preliminary results are encouraging, even without compiling the VM code into native machine code. Moreover, we expect that most of the ideas developed for Athena should carry over to other type-ω DPL’s, allowing for efficient implementations of theorem provers for various different logics.

To ensure logical soundness, Athena performs full Hindley-Milner inference dynamically rather than statically, and a naive implementation will be excessively penalized. A new implementation technique was discovered that achieves optimal performance by postponing work until all redundant computations can be detected and avoided. Also, Athena was extended with a novel technique, based on DPL ideas such as assumption bases, for performing structural induction over arbitrary free algebras. Structural induction is a key tool in the study of programming languages, and this development will enable Athena to be used for proving important properties of programming languages.

8 DOLL subcontract

This subproject serves as the main application area for our DDA efforts. A reflective architecture has been implemented that works by synthesizing code from a specification using modules that include local self tests. The generated code is linked to the specification that generated it and the program synthesis engine is available at runtime. When the base program runs it must execute the pre- and post-tests before invoking the individual modules of the program. As long as the tests succeed the program continues but when a test fails it invokes the next level up the reflective tower where the failure is understood in terms of the specification and the failing module. At that point the applicability of the module is understood to be inappropriate and the synthesis engine can resynthesize taking into account what is now known about the state of the world. This can happen at multiple levels because the task of synthesizing a program from a specification is implemented with a meta program that implements a meta specification. If the synthesizer is unable to synthesize a suitable program, an exception is raised at the next higher level, and so on for as many levels as makes sense for the problem at hand (typically a small number).
The initial version of this architecture was agent-based. The synthesis of code from specifications involves connecting together parts of the solution in the form of modules with entry and exit tests. Instead of having the code generated as a problem-solving activity the solution can be template driven. Templates of successful solution prototypes can be hand generated based on experience. The templates have requirements for how the entries are to be filled and the entries can then be chosen with self tests as before in order to populate the template. When a template cannot be successfully filled a different template must be chosen. The task of populating a template with modules whose signatures match the templates requirements is very similar to method combination.

Instead of methods being chosen purely on the basis of a matching signature, there is a procedure for evaluating the utility of a proposed method. The method combination then evaluates the collection of methods that yield the greatest global utility for the template (we want the template as a whole to operate well).

The templates described above are essentially “super routines”. The protocol for template selection and instantiation involves a protocol for introducing the utility assessment as well as the executing the method pre and post tests. In this model, method recombination occurs when a pre or post test fails. The equivalent to condition handlers in this model is code that is executed to update the utility model, which will then be employed for the recombination phase.

While the template populating approach is, in a sense, less general than the code synthesis approach implemented before, it has a number of attractive features.

1. It allows more explicit control over the behavior of the program because the templates are programmed by hand. This makes it easier to be confident that the behavior of the system will follow a well-understood path.

2. It is cast in the form of a programming model in which program code, self tests, and a utility model each can be represented clearly in the form of program code and debugged in a traditional way.

We documented the protocol for template instantiation, recasting the initial agent-based implementation. The resulting document will be used as the basis for the new implementation of the code in a form that can be distributed so that other groups can experiment with building self-adaptive code.

8.1 The GRAVA Architecture

The work performed for this subcontract was based on a self-adaptive architecture for aerial surveillance (Rob99a,b). GRAVA (for Grounded Reflective Adaptive Vision Architecture) is a self adaptive architecture that segments and labels aerial images in a way that attempts to mimic the competence of a human expert.

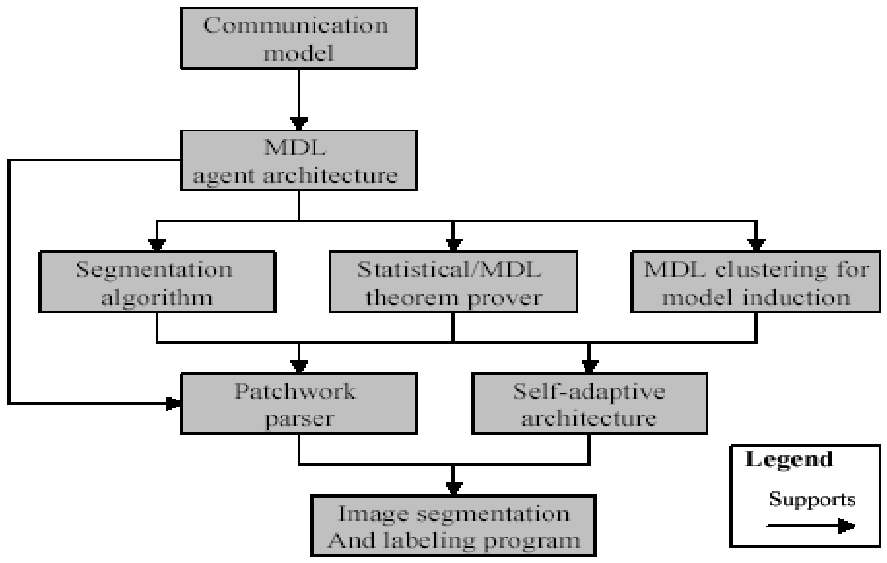
Communication model
MDL agent architecture
Segmentation algorithm
Statistical/MDL theorem prover
MDL clustering for model induction
Patchwork parser
Self-adaptive architecture
Image segmentation And labeling program
Legend
Supports

Figure 5: Logical Components

Figure 5 shows the logical components of the system along with the supporting relationships between the parts. We now sketch the roles of these components.

To produce an image interpretation, a variety of tools need to be brought into play. First, the image is processed by various tools in order to extract texture or feature information. The selection of the right tools determines ultimately how good the resulting interpretation will be. Next, a segmentation algorithm is employed in order to produce regions with outlines whose contents are homogeneous with respect to content as determined by the chosen texture and feature tools. The segmentation algorithm also depends upon tools that select seed points that initialize the segmentation. The choice of tools to initiate the segmentation determines what kind of segmentation will be produced.

Labeling the regions depends upon two processes. The first tries to determine possible designations for the regions by analyzing the pixels within the regions. The second is a statistical parser that attempts to parse the image using a 2D grammar. Our application currently doesn't make use of the parse; but it could be used as the basis for further image interpretation. An important side effect for our application is that contextual information mobilized by the parse process enables good labels to be chosen for regions when there may be several ambiguous possibilities if one only looks at the pixels within the region.

At any point, a bad choice of tool---for initial feature extraction, seed point identification, region identification, or for contextual constraints --- can lead to a poor image interpretation. The earlier the error occurs, the worse the resulting interpretation is likely to be.

The problem of interpreting the real world is inherently ambiguous. A speech or vision program must select the most likely interpretation from the ambiguous candidates. Selecting the most likely interpretation is equivalent to selecting the interpretation with the minimum description length (MDL). We developed apparently for the first time an agent architecture based on the MDL principle, and supporting a conjecture of Leclerc (Lec89) that MDL can apply to higher-level semantics.

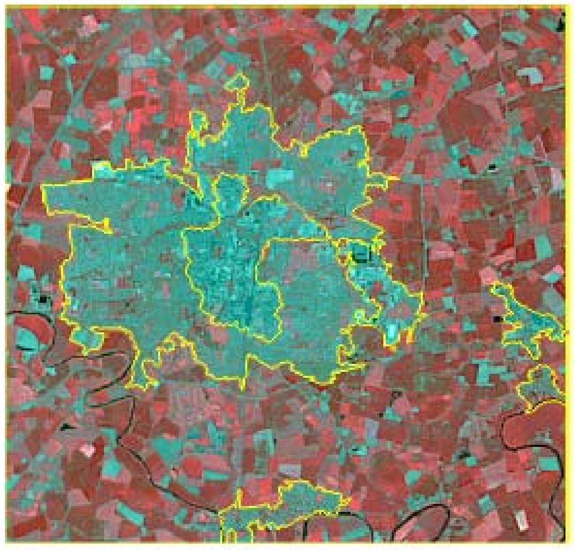

Figure 6: Segmented Image

8.1.1 GRAVA's Self Adaptive Architecture

The goal of the architecture is to support self-adaptation. When the self-assessment determines that the program is doing poorly, the program should seek some way of adjusting its structure so as to do better. The self-adaptive architecture is a collection of supporting capabilities that permits this simple approach to self-adaptation to work. The supporting components are as follows:

1. **Self-assessment**---the ability of a computational agent to evaluate how well it is doing at its current task. The GRAVA architecture provides a protocol for supplying self-assessment functions.

2. **Structure building**---the mechanism that constructs a program from a collection of computational agents. This structure building apparatus is invoked whenever self-assessment indicates poor performance; the system tries to improve by re-synthesizing its program code, using the statistical theorem prover.

3. **Reflection**---the support for self-understanding within the system. By inspecting the state of the embedded semantic account, the system can reason about what the system is doing in terms of a goal that its actions are intended to achieve.

8.2 Debugging and Documenting the GRAVA Architecture

We focused our efforts on debugging and documenting the GRAVA architecture. The documentation will serve as working documentation for the ongoing use of the GRAVA architecture and will be the basis for conference and journal papers. We have debugged and documented the following areas.

8.2.1 Development support

The GRAVA architecture monitors its relationship with the world by running certain tests (pre-tests and post-tests). When they fail an adaptation event occurs which tries to adapt the program to fit the environment better. Pre-tests and post-tests are associated with interpreters. An error system provides support for monitoring when a pre-test or post-test fails.

When an adaptation event occurs a shift in meta-level occurs in the reflective interpreter. We provide the ability to set breakpoints on meta-level transitions so that self-adaptation events can be monitored. Finally, the individual agents can be monitored by break points so that agents can be debugged. We also provide a mechanism for setting breakpoints on the top level return of a solution, so that the selected solution can be interrogated.

In each of the above, the breakpoints can be set for all reflective levels, a specific reflective level, or for a list of reflective levels.

The breakpoints can be set to execute a piece of code rather than "break" so that the software tracing and monitoring of performance can be achieved.

8.2.2 Reflection

The generalized and expanded implementation of GRAVA and the approach to multi-sensor tracking makes use of GRAVA's reflective self-adaptive architecture.

8.2.3 Scarce resources management

Resource management works by:

1. Producing a representation of the resource budget,
2. Producing a representation of the cost of running an agent,
3. Estimating the extent to which agents overlap in their coverage of the interpretation space, and
4. Distributing the resource budget over the entire program.

9 Conclusion

We have been mostly successful in achieving our goals and responding to our challenges. At least with respect to making advanced, functional programming languages useful, practical tools for both serious systems programming and embedded software development, we believe that we have largely been successful. With respect to the goal of "closing the feedback loop" on software construction, we have at least made a credible start. This goal is one that will involve decades of research by many talented research groups before the goal is actually reached, but many improvements have been made, and will be made in the near future.

Throughout the final report we have indicated areas that require further work. We enumerate a subset of those here:

1. Improving the DDA services related to diagnosis, repair, rollback and resourcing.
2. Implementing a translation from the Java programming language to the DVM.
3. Adding features to Java that tap some of the more dynamic aspects of the underlying DVM.
4. Dynamic Optimization Technology
5. Application of DPLs to actual software systems.

10 References

[Ark00] Konstantine Arkoudas. *Denotational Proof Languages*. Ph.D. thesis, Massachusetts Institute of Technology Artificial Intelligence Laboratory, 2000.

[Ark01a] Konstantine Arkoudas. Certified computation. Technical Report AI Memo 2001-007, Massachusetts Institute of Technology Artificial Intelligence Laboratory, 2001.

[Ark01b] Konstantine Arkoudas. Simplifying transformations of type-alpha certificates. Technical Report AI Memo 2001-031, Massachusetts Institute of Technology Artificial Intelligence Laboratory, 2001.

[Ark01c] Konstantine Arkoudas. Type-alpha dpls. Technical Report AI Memo 2001-025, Massachusetts Institute of Technology Artificial Intelligence Laboratory, 2001.

[Ark01d] Konstantine Arkoudas. Type-omega dpls. Technical Report AI Memo 2001-027, Massachusetts Institute of Technology Artificial Intelligence Laboratory, 2001.

[BP01] Jonthan Bachrach and Keith Playford. The java syntactic extender *(jse). In Proceedings of the OOPSLA '01 conference on Object Oriented* Programming Systems Languages and Applications, pages 31–42. ACM Press, 2001.

[CC99] Craig Chambers and Weimin Chen. Efficient multiple and predicate dispatching. In Loren Meissner, editor, *Proceeings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99), volume 34.10 of ACM Sigplan Notices*, pages 238–255, N. Y., November 1–5 1999. ACM Press.

[EKC98] Michael Ernst, Craig Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In Eric Jul, editor, *ECOOP '98—Object-Oriented Programming, volume 1445 of Lecture Notes in Computer Science*, pages 186–211. Springer, 1998.

[FBB+97] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The flux OSKit: A substrate for kernel and language research. In *Symposium on Operating Systems Principles*, pages 38–51, 1997.

[Kic96] Gregor Kiczales. Beyond the black box: Open implementation. *IEEE Software*, January 1996.

[Lan64] P. J. Landin. The mechanical evaluation of expressions. In *Computer Journal*, volume 6, pages 308–329, 1964.

[Lec89] Leclerc, Y. G. 1989. Constructing simple stable descriptions for image partitioning. Int. J. of Computer Vision 3: pp73-102.

[Rob99a] Robertson, P. and Brady, J. M., 1998. Adaptive Image Analysis for Aerial Surveillance. In IEEE Intelligent Systems (!4) #3 May/June pp30-36

[Rob99b] Robertson, P. 1999. A Corpus Based Approach to the Interpretation of Aerial Images. In Proceedings IEE IPA99, IEE (Manchester).

[San79] Erik Sandewall. Why superroutines are better than subroutines. Technical Report LiTH-MAT-R-79-28, Linkoping University, November 1979.

[SCM99] Olin Shivers, James W. Clark, and Roland McGrath. Atomic heap transactions and fine-grain interrupts. In *International Conference on Functional Programming*, pages 48–59, 1999.

[Shr79] Howard Shrobe. Dependency directed reasoning for complex program understanding. Technical Report AI Lab Technical Report 503, MIT Artificial Intelligence Laboratory, April 1979.

[Shi94] Olin Shivers. A scheme shell. Technical Report MIT/LCS/TR-635, Massachusetts Institute of Technology, 1994.

[Shi96a] Olin Shivers. Supporting dynamic languages on the java virtual machine. Technical Report AIM-1576, Massachusetts Institute of Technology Artificial Intelligence Laboratory, 1996.

[Shi96b] Olin Shivers. A universal scripting framework, or lambda: the ultimate 'little language.'. In Joxan Jaffar and Roland H. C. Yap, editors, *Concurrency and Parallelism, Programming, Networking, and Security,* volume 1179 of Lecture Notes in Computer Science, pages 254–265. Springer, 1996.

[Shi97a] O. Shivers. Continuations and threads: Expressing machine concurrency directly in advanced languages, 1997.

[Shi97b] Olin Shivers. Automatic management of operating system resources. In International Conference on Functional Programming, pages 274–279, 1997.

[Shi99] Olin Shivers. Continuations and transducer composition (extended abstract). Technical Report Express 1999-01, Massachusetts Institute of Technology Artificial Intelligence Laboratory, 1999.

[Shi02] Olin Shivers. A simple and efficient natural merge sort., 2002. [Sul01] Gregory T. Sullivan. Dynamic partial evaluation. In Olivier Danvy and Andrzej Filinski, editors, *Programs as Data Objects 2*, volume 2053 of *LNCS*, pages 238–256. Springer-Verlag, May 2001.

[SSS81] Erik Sandewall, Claes Stromberg, and Henrik Sorensen. Software architecture based on communicating residential environments. In *Fifth International Conference on Sofware Engineering*, San Diego, 1981.

[Sul02] Gregory T. Sullivan. Advanced programming language features for executable design patterns. Lab Memo AIM-2002-005, MIT Artificial Intelligence Laboratory, 2002.

[Uck01] Aaron Mark Ucko. Predicate dispatching in the common lisp object system. Technical Report AI Memo 2001-006, Massachusetts Institute of Technology Artificial Intelligence Laboratory, 2001.

11 Appendix

The following papers, each of which represents research sponsored all or in part by DARPA Express project funding, are included by reference.

1. Konstantine Arkoudas. *Denotational Proof Languages*. Ph.D. thesis, Massachusetts Institute of Technology Artificial Intelligence Laboratory, 2000.
2. Konstantine Arkoudas. Certified computation. Technical Report AI Memo 2001-007, Massachusetts Institute of Technology Artificial Intelligence Laboratory, 2001.
3. Konstantine Arkoudas. Simplifying transformations of type-alpha certificates. Technical Report AI Memo 2001-031, Massachusetts Institute of Technology Artificial Intelligence Laboratory, 2001.
4. Konstantine Arkoudas. Type-alpha dpls. Technical Report AI Memo 2001-025, Massachusetts Institute of Technology Artificial Intelligence Laboratory, 2001.
5. Konstantine Arkoudas. Type-omega dpls. Technical Report AI Memo 2001-027, Massachusetts Institute of Technology Artificial Intelligence Laboratory, 2001.
6. Jonthan Bachrach and Keith Playford. The java syntactic extender *(jse). In Proceedings of the OOPSLA '01 conference on Object Oriented* Programming Systems Languages and Applications, pages 31–42. ACM Press, 2001.
7. Olin Shivers, James W. Clark, and Roland McGrath. Atomic heap transactions and fine-grain interrupts. In *International Conference on Functional Programming*, pages 48–59, 1999.
8. Olin Shivers. Supporting dynamic languages on the java virtual machine. Technical Report AIM-1576, Massachusetts Institute of Technology Artificial Intelligence Laboratory, 1996.
9. Olin Shivers. A universal scripting framework, or lambda: the ultimate 'little language.'. In Joxan Jaffar and Roland H. C. Yap, editors, *Concurrency and Parallelism, Programming, Networking, and Security,* volume 1179 of Lecture Notes in Computer Science, pages 254–265. Springer, 1996.
10. O. Shivers. Continuations and threads: Expressing machine concurrency directly in advanced languages, 1997.
11. Olin Shivers. Automatic management of operating system resources. In International Conference on Functional Programming, pages 274–279, 1997.
12. [Shi99] Olin Shivers. Continuations and transducer composition (extended abstract). Technical Report Express 1999-01, Massachusetts Institute of Technology Artificial Intelligence Laboratory, 1999.

13. Olin Shivers. A simple and efficient natural merge sort., 2002. [Sul01] Gregory T. Sullivan. Dynamic partial evaluation. In Olivier Danvy and Andrzej Filinski, editors, *Programs as Data Objects 2*, volume 2053 of *LNCS*, pages 238–256. Springer-Verlag, May 2001.

14. Gregory T. Sullivan. Advanced programming language features for executable design patterns. Lab Memo AIM-2002-005, MIT Artificial Intelligence Laboratory, 2002.

15. Aaron Mark Ucko. Predicate dispatching in the common lisp object system. Technical Report AI Memo 2001-006, Massachusetts Institute of Technology Artificial Intelligence Laboratory, 2001.